B5 – Advanced Functional Programming

B-FUN-500

GLaDOS

Generic Language and Data Operand Syntax

{EPITECH.}

GLaDOS

binary name: glados
language: haskell
compilation: via Makefile, including re, clean and fclean rules
build tool: free (stack recommanded)

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

The goal of this project is to implement a programming language of your own design, in Haskell. Don't panic just yet, we will go there step by step.

The project will be split in different parts that are all mandatory. Read them with attention, they are designed to help you progress on a smooth difficulty curve.

Also for your trouble, cake and grief counseling will be available at the conclusion of the test.

{ EPITECH.}

Part 0 : The Enrichment Testing Center

> The Enrichment Center promises to always provide a safe testing environment. In dangerous testing environments, the Enrichment Center promises to always provide useful advice. For instance: division by zero will kill your program. Try to avoid it.

Tests are not only useful, they prevent you from shooting yourself in the foot when adding more code.
To be sure you develop a perfectly fine language, the GLaDOS system requires you to test **thouroughly** your code.

Build system and dependencies

You are free to use any build system you see fit, but you **must** use one (**stack** is always a safe choice). During each defense you will be asked to demonstrate how to build your project from scratch (i.e from a fresh git clone).

Forbiden dependencies

You have a fair amount of freedom in the modules you are allowed to use, but there is a big no-no:
NO PARSING LIBRARY (neither as an external module, nor included in the base distribution), you have to make your parsing yourself from scratch, otherwise you will be flaged as cheaters.

Unit & Integration tests

You are free to use any unit testing framework you want, but you need one and you **must** have a comprehensive testing policy, including unit and integration test suits.
You **must** be able to show how much of your code is covered by tests.

Continuous Integration and Continious Delivery

Finally, to end this part on the wonders of testing, we'll ask you to automate everything by building a CI / CD.
You must choose the CI of your choice to automate the process of testing.
A good idea for example would be to test each part of the code **before** each push to prevent from pushing bad code.

To satisfy the CD part of the requirement, your system must produce a fully functionnal release build automaticaly.

> **Part 0 is absolutely mandatory for every defense**
> Failure to comply will result in an 'unsatisfactory' mark on your official testing record followed by death (of your program but still).

Part 1 : Lots of Irritating Superfluous Parentheses (LISP)

For this first part of the project you **must** implement a minimalist **LISP** interpreter. To be more precise, when not specified otherwise in this document, your language must behave just like Chez-Scheme. It will be evaluated during the first Defense of this project.

In the rare occasion of you being one of those humans who think he's faster than everyone else, and you are already at part 2 before even the first defense, you **MUST** still have the LISP clone code available (on a separate branch of your git, or as an option of your latest build).

Failure to do so will result in you **not having cake** at your first defense, nor points nor credits either.
Also, be sure to read the following warning. Repetition is key.

> **Part 0 is absolutely mandatory for every defense**
> Failure to comply will result in an 'unsatisfactory' mark on your official testing record.

Syntax

Being a LISP, this first language **must** be represented by Symbolic-Expressions. At the bare minimum your parser **must** be able to handle:

- Atoms:

    - Signed Integers in base 10
    - Symbols (any string which is not a number)

- Lists:

    - started by en open parenthesis, ended by a close parenthesis
    - contains zero, one or any number of sub-expressions separated by spaces, tabs, carriage returns or parenthesis.

Examples of valid S-Expressions:

```
foo

42

(1 2 3)

((foo bar baz)
    (1 2 3) ()
    (((((1(2)3)))))
  )
```

INVOCATION AND ERROR HANDLING

Your program **must** be able to read your language code from standard input. You are free to add more and fancier way to invoke it (from files given as arguments, with a full featured REPL, etc.).

```
                            Terminal                         –  +  x
~/B-FUN-500> cat foo.scm
(define foo 21)
(* foo 2)
~/B-FUN-500> ./glados < foo.scm
42
```

You must stop the execution as soon as an error occurs and return a 84 status code. You're free to display any meaningful information on the standard output or error output.

```
                            Terminal                         –  +  x
~/B-FUN-500> cat error.scm
(* foo 2)
~/B-FUN-500> ./glados < error.scm
*** ERROR : variable foo is not bound.
~/B-FUN-500> echo $?
84
```

CORE CONCEPTS

At the bare minimum your program **must** handle the following concepts:

TYPES

- You language **must** support 64 bit integers and boolean values.
- Boolean values **must** be represented by the "**#t**" and "**#f**" symbols, for True and False respectively.
- As a consequence of being a functional language, it **must** also support a procedure type (more information about that bellow))

Optionally, you are free to implement more types (for example: lists).

BINDINGS

You **must** implement a way to **define** an association (binding) between a symbol and a value (which can be seen as having variables, to be able to store them and reuse them afterwards). When a symbol is bound to a value, it evaluates to this value. Trying to evaluate an unbound symbol produces an error.

The **define** notation is a special form which binds an expression to a symbol:

```
(define <SYMBOL> <EXPRESSION>)
```

```
> foo
*** ERROR : variable foo is not bound.
> (define foo 42)
> foo
42
```

USER DEFINED FUNCTIONS AND LAMBDA

You **must** define a way to represent (and call) functions, being anonymous ones (**lambdas**) or named ones. Functions **must** be able to take parameters (or none).

A function call (application) is simply a list with the callee (the operator) placed in first place, the other elements of the list being the arguments (the operands). This is the default behavior when a list doesn't match with a special form.

```
~/B-FUN-500> cat call.scm
(div 10 2)
~/B-FUN-500> ./glados call.scm
5
```

Lambdas

A lambda is a special form composed of a (possibly empty) list of parameters and a body.
The body is an expression, which will be evaluated when the lambda is called, within a context where the parameters will take the values of the arguments provided during invocation.

A lambda has the following form:

```
(lambda (<ARG1> <ARG2> ... <ARGN>) <BODY>)
```

```
~/B-FUN-500> cat lambda1.scm
(lambda (a b) (+ a b))
~/B-FUN-500> ./glados < lambda1.scm
#<procedure>
~/B-FUN-500> cat lambda2.scm
((lambda (a b) (+ a b)) 1 2)
~/B-FUN-500> ./glados < lambda2.scm
3
~/B-FUN-500> cat lambda3.scm
(define add
   (lambda (a b)
      (+ a b)))
(add 3 4)
~/B-FUN-500> ./glados < lambda3.scm
7
```

Named functions

In this language named functions are just syntactic sugar added to the **define** notation. The following example produces the same result as **"lambda3.scm"** above:

```
(define (<FUNC_NAME> <ARG1> <ARG2> ... <ARGN>) <BODY>)
```

```
~/B-FUN-500> cat function1.scm
(define (add a b)
   (+ a b))
(add 3 4)
~/B-FUN-500> ./glados < function1.scm
7
```

> Named functions **must** be capable to call themselves, to allow recursion.

CONDITIONAL EXPRESSIONS

You **must** define a way to represent conditions using a **if** notation, which contains a conditional expression followed by two more expressions. The first of these expressions is evaluated and returned if the condition is true, otherwise the second is evaluated and returned.

It **must** have the form:

```
(if <CONDITION> <THEN> <ELSE>)
```

Where CONDITION, THEN and ELSE are three arbitrarily complex sub-expressions.

```
~/B-FUN-500> cat if1.scm
(if #t 1 2)
~/B-FUN-500> ./glados < if1.scm
1
~/B-FUN-500> cat if2.scm
(if #f 1 2)
~/B-FUN-500> ./glados < if2.scm
2
~/B-FUN-500> cat if3.scm
(define foo 42)
(if (< foo 10)
   (* foo 3)
   (div foo 2))
~/B-FUN-500> ./glados < if3.scm
21
```

> You are free to implement the **cond** notation too but it's not mandatory.

BUILTIN FUNCTIONS

You've seen some of them already in the examples, but in order to make your language (barelly) usable, you **must** implement some functions which will be hardcoded in your interpreter:

- Predicates, which take two arguments and evaluates to a boolean value:

    - **"eq?"** (returns true if its two arguments are equal, false otherwise)
    - **"<"** (returns true if the first argument is strictly lower than the second, false otherwise)

- Aritmethic operators, which take two arguments and return an integer:

    - **"+"**, **"-"**, **"*"**
    - **"div"** (division) and **"mod"** (modulo)

> Even if they are represented by a single special character, they are just symbols bound to a function and behave like any user defined functions.

```
~/B-FUN-500> cat builtins1.scm
(+ (* 2 3) (div 10 2))
~/B-FUN-500> ./glados < builtins1.scm
11
~/B-FUN-500> cat builtins2.scm
(eq? (* 2 5) (- 11 1))
~/B-FUN-500> ./glados < builtins2.scm
#t
~/B-FUN-500> cat builtins3.scm
(< 1 (mod 10 3))
~/B-FUN-500> ./glados < builtins2.scm
#f
```

EXAMPLES

Given all the rules above, Your lisp interpreter should be able, for example, to process the following programs:

```
~/B-FUN-500> cat superior.scm
(define (> a b)
   (if (eq? a b)
      #f
      (if (< a b)
         #f
         #t)))
(> 10 -2)
~/B-FUN-500> ./glados < superior.scm
#t
~/B-FUN-500> cat factorial.scm
(define (fact x)
  (if (eq? x 1)
      1
      (* x (fact (- x 1)))))
(fact 10)
~/B-FUN-500> ./glados < factorial.scm
3628800
```

Part 2 : embrace and extend

At this point you should have a minimal but functional core language.
Your goal for this second and last part is to make it evolve into a more advanced system on two axis:

- Its syntax, grammar and semantics (skill: **parsing**)
- The way and how fast it executes code (skill: **evaluation / compilation**)

> To reach a good grade your project must feature improvements on each of these axis compared to Part 1.

> Additionally, to reach the maximum level of credits (bonuses), your project must also implement more general features (types supported, data structures, input/outputs, FFI, etc.)

Syntax, grammar and semantics

Symbolic expressions are neat but all these parenthesis can be hard to parse for a human eye. Ditch your S-Expression parser and implement your own grammar (or keep the S-Expression front-end and add a new one).

You have a lot of freedowm in the form you can give to your language. Nevertheless, you still have to follow some rules, and you're advised to not try to reinvent the wheel: you're encouraged to get your inspiration from the languages you already know.

Your language can't be line oriented. We ask you to implement a high level language, not a glorified assembler. While some carriage returns may be mandatory in your grammar, any expression must be splitable on several lines.

During the final defense we will look for several features of your language, implement them to get a better grade:

- A formal document describing your language's grammar (you're encouraged to use **BNF**). **This part is mandatory to validate**.
- A consistent, unclutered grammar. for example:
    - if your language generates more parenthesis than LISP, maybe you took a bad turn somewhere...
    - if your language use curly brackets and explicit return statement (just like C) but is otherwise not imperative, maybe there's a more concise way to represent your code.
- Some syntactic sugar (two (or more) syntactic forms which result in the same AST)
- Infix arythmetic operators with the expected priorities (remember the EvalExpr?)
- Infix user defined operators with user defined priority (as in Haskell or Kaleidoscope for example)

EVALUATION AND COMPILATION

If you went for the simplest solution in the first part, you should now have a basic AST walking, environment passing interpreter. It's fine for a first try but you can do better.

All of the following solutions are suitable goals for your project, but some of them are more complex to implement, therefore they will be valued more during the final defense. Still, it's always better to have a simpler, fully functional system than an advanced but broken one.

In any case, all these solutions involve a form of compilation (even the ones named "interpreter"). We consider as compilation the action of taking a data structure in a form of a tree, and to convert it to a data structure in a form of a list (of instructions). According to this principle, "transpiling" from one high level language to another won't satify this part.

- Advanced interpreter:

  - Stack based
  - Registers based
  - both

- Virtual machine:

  - instructions written to a file in binary form (bytecode)
  - loaded and executed by another (haskell) program

- Binary objects (".o" files) targeting existing architectures (x86_64, arm, etc.)

  - using LLVM bindings for haskell (llv and llvm-pure modules)
  - using your own code generator to produce native machine code

> The different options listed above are presented in incrementaly difficult order. Some of them are natural precursor to the next level, some others are on a different track altogether

Bonus

In order to unlock the maximum level of credits, you must implement more features in your language. We give you a lot of freedom here, but they must correspond to one of the main "tracks" defined here:

- More data types:

    - Floating point numbers (with corresponding division operator: /)
    - Symbols as data (as in LISP using quote, or as in Erlang/Elixir, or Prolog)
    - Lists
        - with corresponding builtins to manipulate them: cons/car/cdr or :/head/tail
        - syntactic sugar for literals, for example: [1,2,3]
    - Strings (as lists of Char or with a custom set of builtins to use them)
    - Tuples, Structures, Arrays, HashMaps? etc.

- Imperative constructs:

    - capability to describe functions as sequence of statements
    - mutable variables
    - loops

- Side effects:

    - add builtins or a notation to read or write to standard input / output
    - add builtins or a notation for file handling
    - bindings to graphical library, or networking primitives, etc.
    - a notation to interface with any external function (Foreign Function Interface)

- Type inference:

    - being able to detect before execution if a program / function is not properly typed
    - provide a notation to anotate function / expressions with types

- Metaprogrammation:

    - provide notations to allow the user to programmaticaly modify the program structure
    - for example: LISP macros, Rust macros, Elixir macros…

- Optimisation

    - TCO (tail call optimisation)
    - anything that make your code run faster (don't forget the benchmarks to prove it!)

Resources

- Related to **Part 1**:
    - The Roots of Lisp
- Related to **Part 2**:
    - 500 Lines or Less | A Python Interpreter Written in Python